**Phoebe Nichols, pmn29**

**Project Documentation**

## 1 Lexer and Parser

The lexer is generated using `ocamllex`, and the parser is generated using `menhir`.

The grammar is:

```
Program -> Sentence ?- Resolvant | Sentence
Resolvent -> ClauseBody
Sentence -> Clause Sentence | Clause
Clause -> Atom :- ClauseBody. | Atom
ClauseBody -> Atom | IsExpr | Atom , ClauseBody | IsExpr , ClauseBody
IsExpr -> variable is Arith
Arith -> Arith + Arithbase | Arithbase
Arithbase -> int | Variable | - int
Atom -> name ( TermList ) | name
TermList -> Term | Term, TermList
Term -> variable | - int | int | name | name ( TermList )
```

```
variable is ['A'-'Z']['a'-'z''A'-'Z''0'-'9']*
int is ['0'-'9']+
name is ['a'-'z']['a'-'z''A'-'Z''0'-'9']*
```

All white-space is ignored.

## 2 Interpreter

The interpreter takes the parse tree and generates a hash map from clause id (name * arity) to a list of clauses. A goal atom is proved by fetching all the clauses for the predicate called by the atom. The first of these clauses is selected to attempt to prove the goal atom. Upon failure, the next clause is tried (this is backtracking).

A substitution is represented by a hash map from term to term. The problem of alpha renaming is avoided by having the unification of two terms return two distinct substitutions. The substitution for a term contains values to replace only variables in that term. This enables me to keep track of the context in which each substitution is relevant. The interpreter initially used functions to represent substitutions: the representation of a substitution was just a function that would perform the substitution.

The interpreter maintains a stack of choice points for backtracking. A choice point contains a goal atom, a list of remaining clauses, and a substitution. The saved substitution is needed to 'translate' a substitution proving the goal atom so it is relevant in the context to which it is returned.

[A lot of this is unclear, but I'm not sure if explaining it properly is worth it because it's just a preliminary thing]

## 3 Abstract Machine

The abstract machine uses a fully structure sharing implementation. This is different to the WAM, which uses structure copying for building terms but uses structure sharing for the goals in a resolvant.

The instruction set is:

```
type instruction =
    TryMeElse of clausePos
  | RetryMeElse of clausePos
  | TrustMe
  | GetVariable of env * arg
  | GetValue of env * arg
  | GetStructureA of structure * arg
  | GetStructureT of structure * temp
  | GetIntA of arg * int
  | GetIntT of arg * int
  | UnifyVariableE of env
  | UnifyVariableT of temp
  | UnifyValueT of temp
  | UnifyValueE of env
  | UnifyInt of int
  | PutStructureA of structure * arg
  | PutStructureT of structure * temp
  | PutIntA of arg * int
  | PutIntT of temp * int
  | PutVariable of env * arg
  | PutValue of env * arg
  | SetVariableE of env
  | SetValueE of env
  | SetIntE of int
  | SetValueT of temp
  | Allocate of int
  | Deallocate
  | Proceed
  | InitAcc of temp * env
  | AddE  of temp * env
  | AddI of temp * int
  | Is of env * temp
  | Call of functionId
  | Return
  | Empty
```

Example bytecode is:

## 4 Initial performance measurements